A Gentle Introduction to UVM

Ryan Billmeyer

June 17, 2022

Abstract

The notes are adapted from The UVM Primer by Ray Salemi.

1 Introduction

These notes cover a working introduction to the Universal Verification Methodology (UVM). The goal is to be familiar enough with the UVM to interface with and use it. Sections of code are stylized as

```
module top;
    tinyALU(.*);
endmodule
```

2 The Conventional Testbench

UVM fundamentally builds off the SystemVerilog testbench architecture, so we will start there. The goal of our testbench is to implement a "coverage-first" methodology by

- Exhaustively testing the DUT functionality
- Exercising all lines of RTL
- Exercising all datapaths through the DUT

In addition, our testbench should be self-checking so we don't have to sift through the results.

2.1 TinyALU

The design that will be referenced throughout these notes is called TinyALU. As the name implies, this is a tiny arithmetic logic unit. It takes in two 16-bit operands, a 3 bit op-code, and a start bit. TinyALU outputs a 16-bit result and a done signal. As a verbal timing diagram: The start signal is asserted high on the posedge of the clk, the operands and op-code are read in, the operation is performed in N cycles, the result is calculated and the done signal is raised, then the start signal is lowered. The design is reset through a synchronous active low reset line. It supports 5 operations: NOP (3'b000), ADD (3'b001), AND (3'b010), XOR (3'b011), MULT (3'b100).

2.2 TinyALU Testbench

The components of the simple SV testbench include a top level declaration, a coverage block with covergroups, a monitor, a driver, a generator and a scoreboard. The coverage block measures what functionality we have tested. The scoreboard achieves self-checking. The generator creates constrained random stimulus. This architecture is discussed more in-depth in SystemVerilog for Verification. We can expand up by increasing the reusability of each of these pieces.

3 Interfaces and Bus Functional Models

The ultimate targets for any verification environment are modularity and abstraction. An important concept towards these goals is the SystemVerilog interface. Interfaces encapsulate the port signals of a module or object, greatly simplify how we connect to them. Interfaces can then be used to encapsulate bus protocols into simple routines through Bus Functional Models (BFM).

3.1 Bus Functional Model (BFM)

A BFM is an interface. For the TinyALU, our BFM will contain all the signals needed, provide a clock, a reset task, and a send op task.

```
interface tinyalu_bfm ();
    byte unsigned A;
    byte unsigned B;
    logic clk;
    logic reset_n;
    logic [2:0] op;
    logic start;
    logic done;
    logic [15:0] result;

    modport dest(
        input A,
        input B,
        input clk,
        input reset_n,
        input op,
        input start,
        output done,
        output result
    );

    modport src(
        output A,
        output B,
        input clk,
        output reset_n,
        output op,
        output start,
        input done,
        input result
    );

    task gen_clk();
        clk = 0;
        forever begin
            #1000
            clk = ~clk;
        end
    endtask : gen_clk

    task reset_alu();
        reset_n = 1'b0;
        @(negedge clk);
        @(negedge clk);
        reset_n = 1'b1;
```

```
        start   = 1'b0;
    endtask : reset_alu

    //Defines protocol to interact with DUT given op
    task send_op(input byte iA, iB, input logic [2:0] iop, output var logic [15:0] resul
        op = iop;
        if(iop == rst_op) begin
            @(posedge clk);
            reset_alu();
        end else begin
            @(negedge clk);
            A = iA;
            B = iB;
            start = 1'b1;
            if (iop == no_op) begin
                @(posedge clk);
                #1;
                start = 1'b0;
            end else begin
                do
                    @(negedge clk);
                while(done == 0);
                start = 1'b0;
            end
        end
    endtask : send_op

endinterface : tinyalu_bfm
```

Placing our bus protocol within our BFM allows us to instantiate this interface anywhere we need to drive our TinyALU from a testbench. In addition, if our protocol is updated, we only need to update the send op task in our BFM to update all testbenches.

3.2 Modularizing the Testbench with BFMs

The definition of our BFM allows us to now simply connect the pieces of our verification environment.

```
module top;
    tiny_alu_bfm bfm();
    random_tester random_tester_i (bfm.src);
    coverage coverage_i (bfm.dest);
    scoreboard scoreboard_i (bfm.dest);

    tiny_alu DUT (bfm.dest);
endmodule
```

We can now utilize the tasks defined in the BFM to simplify the elements of our verification environment.

```
module scoreboard(tiny_alu_bfm bfm);
    always @(posedge bfm.done) begin
        logic [15:0] pred_result;
        case (bfm.op)
            add_op: pred_result = bfm.A + bfm.B;
            ...
        endcase

        if((bfm.op != no_op) && (bfm.op_set != rst_op))
            if(pred_result != bfm.result)
```

```
                //report failure
    end
endmodule : scoreboard
```

In the same sense the tester can also be simplified along the lines:

```
module tester (tiny_alu_bfm bfm);
    initial begin
        bfm.reset_alu();
        repeat(1000) begin : random_loop
            op = get_op();
            iB = get_data();
            iA = get_data();
            bfm.send_op(iA,iB, op, result);
        end : random_loop
        $finish
    end
endmodule
```

4 Brief Notes on OOP in SystemVerilog

These topics are fairly basic. I'm highlighting some areas that may be tricky.

4.1 Constructor Inheritance

An extended class must have a constructor defined if an argument is passed into the constructor at instantiation. This means that even if we just intend to call the super new(), we still need to define new() in the extended class if new requires an argument. This is the common case. As a side note, the compiler will be able to tell what constructor you are calling based on the reference type of the object on the LHS.

4.2 Memory allocation for classes

A class object must be instantiated. First, we declare the handle for the class object. Then, we instantiate the object and store the instantiation in the handle.

```
class class_i; //Creating handle
class_i = new(); //instantiate the object and assign to handle
```

Memory for the handle is allocated once it is created. Memory for the non-static class object is allocated at instantiation. Passing the object between environment components is the same as passing the memory space between threads, hence why mailboxes are used for synchronization.

4.3 Virtual and Abstract

OOP allows us to store an extended class object in a base class object. However, we run into issues with what methods are we referring to.

Virtual methods solve these issues and are common to all OOP languages. My favorite definition of a virtual method is a method where the definition depends on the runtime type of the underlying object. On the other hand, a non-virtual method is a method where the definition depends on the reference type of the object at the point of invocation. This means that if an extended class object is instantiated and assigned to a base class object:

- All invoked virtual methods will reference the method definition that is associated with the class type we instantiated and not the one we assigned the object to.
- All invoked non-virtual methods will reference the method definition that is associated with the class type the object is assigned to at the point of invocation.

In essence, a virtual method says that the definition lies somewhere else. An overridden virtual method will inherit the virtual property, so the virtual keyword is not necessary in the extended class.

```
class base_class;
    int a;
    function new(int a);
        this.a = a;
    endfunction : new

    virtual function void method1();
    endfunction : method1
endclass : base_class

class ext_class extends base_class;
    int a;
    function new(int a);
        super.new(a);
    endfunction : new

    //Inherits virtual keyword
    function void method1();
        //override base definition
    endfunction : method1
endclass : base_class

module tb;
    base_class base_class_h;
    ext_class ext_class_h;
    ext_class_h = new(); //instantiate extended class
    base_class_h = ext_class_h; //assign extended class object to base class handle

    If method1 is virtual:
        base_class_h.method1(); //this will invoke method1 definition in ext_class_h
    If method1 is not virtual:
        base_class_h.method1(); //this will invoke method1 definition in base_class_h

endmodule
```

We can take this one step further with the notion of abstract classes and pure virtual methods. These two concepts can be enforced to remove the ambiguity of what method definition we are referring to. Abstract classes can only be base classes; they cannot be instantiated. The methods of an abstract class can be defined as pure virtual methods. Pure virtual methods only specific the declaration of the method. The method definition **must** be provided by extended classes. Abstract classes may also contain virtual or non-virtual methods.

```
//Define an abstract class
virtual class abs_class;
    int a;

    function new (int a);
        this.a = a;
    endfunction

    //Only provides declaration of method
    pure virtual function void method1();

endclass : abs_class
```

```
class ext_class extends abs_class;
    int a;

    function new (int a);
        super.new(a);
    endfunction

    //Override method1 definition
    function void method1();
    endfunction
endclass : ext_class
```

4.4 Static

The static keyword can be applied to variables and methods. It is an intelligent approach to the tricky subject of global resources.

A static data member of a class can be thought of as a global variable that is shared between instances of that class. No matter how many times you instantiate the class, you will always be pointing to the same memory location. As such, the memory for a static data member of a class is allocated at the time the class handle is created. Additionally, the static member can be accessed without instantiating the class. For example, say we want a queue of objects that visible from anywhere in the testbench.

```
class transaction_queue;
    static transaction q[$]; \\queue in SV
endclass

module top;
    transaction transaction_h;
    transaction_h = new(...);
    transaction_queue::q.push_back(transaction_h);
endmodule
```

We extend on this with static methods that can restrict the operations that can be performed on our static data members. The protected keyword means this data member can only be accessed through the class methods, not limited to static cases.

```
class transaction_queue;
    protected static transaction q[$]; \\queue in SV
    static function void push (transaction tran);
        this.q.push_back(tran);
    endfunction
endclass

module top;
    transaction transaction_h;
    transaction_h = new(...);
    transaction_queue::push(transaction_h);
endmodule
```

4.5 Parameterized Class Definitions

The ability to parameterize classes is a natural extension of the HDL ability to parameterize IP blocks. In addition to data variable parameters, we can create type parameters for a class. This allows us to specify the reference type we are operating on at invocation. Take the above static method example. We can enhance our transaction queue class to work with any object type by adding a type parameter.

```
class base_queue #(type T);
    protected static T q[$]; \\queue of type T objects in SV
    static function void push (T obj);
        this.q.push_back(obj);
    endfunction
endclass : base_queue

module top;
    transaction transaction_h;
    driver      driver_h;
    transaction_h = new(...);
    driver_h = new(...);
    base_queue #(transaction)::push(transaction_h); \\specify ref type of trans
    base_queue #(driver)::push(driver_h);
endmodule
```

It is important to note that specifying a different parameter value will create two copies of the base queue: one for transaction type objects and one for driver type objects. By changing the parameter, we are creating a completely new class with that parameter. The classes do not share namespaces.

While the static keyword is useful, it is not always appropriate. Thus, we often use parameterized classes to declare dynamic variables within a defined scope. Let's show this by translating the above example into non-static terms.

```
class base_queue #(type T);
    protected T q[$]; \\queue of type T objects in SV
    function void push (T obj);
        this.q.push_back(obj);
    endfunction
endclass : base_queue

module top;
    transaction transaction_h;
    driver      driver_h;
    base_queue #(transaction) trans_q;
    base_queue #(driver) dvr_q;
    initial begin
        dvr_q = new();
        trans_q = new();
        //invoke methods and stuff
    end
endmodule
```

The instantiated queues now only exist within the scope of module top, not the entire testbench. The memory they occupy will also be de-allocated upon completion of the task.

5 The Factory Pattern

Up until this point, we have been hard-coding the instantiation of our classes using new(). This is bad. We very quickly run into flexibility issues with this approach. What if we want to change the transaction and driver objects to bicycle and car objects? We would have to edit the source code and recompile, wasting away the best years of our life. We can avoid this issue by using the factory pattern. As the name implies, we want to mimic a factory that produces objects. The factory takes in raw materials (constructor arguments) and uses a blueprint (constructor) to output a product (a class object of the desired type). The benefit of this approach is that we can swap the blueprint for a different blueprint (constructor for a different class extended from the same base class) at any time and get our desired output. Swapping the blueprints is handled automatically by our OOP approach; all we need to do is alter the constructor argument. That's a very power technique; the ability to generate a specific object at run time by changing a parameter is not to be understated.

5.1 Creating a Factory

The factory is implemented, unsurprisingly, as a class. There is a static method that is passed the arguments we are interested in and the type of object we wish to create. Given these, the method returns the class object.

```
class car;
    int length, width;
    function new (int length, width);
        this.length = length;
        this.width  = width;
    endfunction

    pure virtual function void drive();

endclass : car

class sedan extends car;
    int avg_mpg;
    function new(int length, width, mpg);
        super.new(length,width)
        this.avg_mpg = mpg;
    endfunction

    function void drive();
        //automatic
    endfunction
endclass

class suv extends car;
    int avg_mpg;
    function new(int length, width, mpg);
        super.new(length,width)
        this.avg_mpg = mpg;
    endfunction

    function void drive();
        //manual
    endfunction
endclass

class car_factory;
    static function car make_car(string model, int length, width);
        suv suv_h;
        sedan sedan_h;
        case (model)
            "sedan" : begin
                sedan_h = new()
                return sedan_h;
            end
            "suv" : begin
                suv_h = new()
                return suv_h;
            end
            default begin
                $fatal();
            end
        endcase
```

```
    endfunction
endclass
```

Thanks to polymorphism, we can return the sedan object as a car type, allowing us to dynamically create different kinds of cars.

5.2 Using the Factory

Since the make car function is static, it can be invoked anywhere within the testbench. The use of a pure virtual drive method allows the invocation of the drive method to correspond to the correct reference type.

There may arise an issue with returning these derived classes as a base class. Specifically, if we tried to access the avg mpg member of the sedan class, we would get an error using the above method because avg mpg is not a member of the car base class. To remedy this, we need to cast the returned car as a sedan.

```
car car_h;
car_h = car_factory::make_car("sedan",1,2);
car_h.drive(); //invoke sedan drive function
cast_ok = $cast(sedan_h,car_h); //cast car as sedan
```

The cast function requires that the class being casted to is a child class of the class we are casting from. We can condense all of this into a single line.

```
if(!$cast(sedan_h,car_factory::make_car("sedan",1,2)))
    $fatal();
base_queue #(sedan)::push(sedan_h);
```

The ability of the factory will be useful to dynamically create our testbench components.

6 An Object-Oriented Testbench

We now wish to transition our module-based testbench towards an object-oriented one. The first pass at an object-oriented testbench for the TinyALU will consist of one module and four classes:

- Top - Top-level module that instantiates the testbench class
- Testbench - Top-level class
- Tester - Drives the stimulus
- Scoreboard - Checks observed results against predicted results
- Coverage - Captures functional coverage information

6.1 Top-level module

The top-level module is mainly responsible for doing three things:

- Importing class definitions
- Instantiating the DUT and BFM and declaring the testbench handle
- Instantiating and executing the testbench class

These objectives are fairly self-explanatory and best showcased via writing out the top module.

```
module top;
    //1. Import class definitions + macros
    import tinyalu_pkg::*;
    `include "tinyalu_macros.svh"

    //2. Instantiate the DUT, BFM
```

```
    tinyalu DUT (bfm.dest);
    tinyalu_bfm bfm();

    //2. Declare testbench handle
    testbench testbench_h;

    //3. Instantiate and execute testbench
    initial begin
        testbench_h = new(bfm);
        testbench_h.execute();
    end
endmodule
```

6.2 Testbench Class

The testbench class is the top level of the testbench hierarchy. The responsibility of the testbench class is to instantiate, connect, and evoke methods from the other 3 classes in our verification environment. The UVM will have written this already. Once again, this is clearest by giving code first.

```
class testbench;
    virtual tinyalu_bfm bfm;

    tester tester_h;
    scoreboard scoreboard_h;
    coverage coverage_h;

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
    endfunction

    task execute();
        //instantiate and connect helper classes
        tester_h = new(this.bfm);
        scoreboard_h = new(this.bfm);
        coverage_h = new(this.bfm);

        //execute classes
        fork
            tester_h.execute();
            scoreboard_h.execute();
            coverage_h.execute();
        join_none
    endtask : execute
endclass : testbench
```

There are two new things in this piece of code that are worth noting.

The first is the use of the keyword **virtual** on the bus functional model interface. This keyword allows us to pass the interface into a class. The keyword virtual is saying that the we will be passing the variable bfm a handle to an interface at some point, but not at compile time (modules are not dynamic objects, which explains why we don't use virtual interfaces with them). We've essentially told the compiler that we got this part, so now we need to actually keep our promise. The approach we take in the above example is simply passing the handle through the constructor, new(bfm).

The second new thing is the fork/join_none. This is how SystemVerilog implements multi-threading. In this case, each line starts a new execution thread and the code will not wait for any of the threads to finish before continuing execution of the parent thread. Other options include fork/join_all, which waits for all threads to complete, and fork/join_any, which waits for any thread to complete.

6.3 Tester Class

This class is called the tester class in the UVM primer. It is also commonly split into two classes called the driver (drives pin stimulus) and the generator (generates the random stimulus).

```
class tester;
    virtual tinyalu_bfm bfm;

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
    endfunction

    task execute();
        repeat(N) begin:
            //generate stimulus
            //drive stimulus
        end
        $finish()
    endtask : execute
endclass : tester
```

6.4 Scoreboard Class

The scoreboard class will observe the result from the DUT, calculate its own result, and compare the two. The scoreboard in the UVM primer combines the function of what is commonly called a monitor (translates DUT output into result) and a scoreboard (predicts result and self-checks).

```
class scoreboard;
    virtual tinyalu_bfm bfm;

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
    endfunction

    task execute();
        forever begin : self_checker
            @(posedge bfm.done)
            //record output
            //predict output
            //self check
        end : self_checker
    endtask : execute
endclass : scoreboard
```

6.5 Coverage Class

The coverage class is also usually incorporated into a scoreboard.

```
class coverage;
    virtual tinyalu_bfm bfm;
    //define covergroup

    function new (virtual tinyalu_bfm bfm);
        this.bfm = bfm;
        covergroup_1 = new();
        covergroup_2 = new();
    endfunction : new
```

```
    task execute();
        //update covergroups
    endtask : execute
endclass : coverage
```

7 UVM Tests

Now the fun begins. We can officially start diving into the UVM. The main driving force behind the UVM is to systematically enable a dynamic testbench. This is a fancy way of saying we want to run many tests with one compile. The object-oriented testbench that was covered in the last section is still hard-coded in the eyes of a UVM purist. We will now dive into how the UVM constructs a dynamic testbench.

7.1 Creating Tests with the Factory Pattern

The UVM is equipped with its own factory that allows us to build almost anything, which means we can create dynamic testbenches. To start, we want to use the factory to launch different tests.

```
vsim testbench -coverage +UVM_TESTNAME=add_test
vsim testbench -coverage +UVM_TESTNAME=random_test
```

In the above, we are compiling the testbench once then supplying two different test names at run time to execute each test. We define these test names as classes then call the UVM factory with the test names using UVM_TESTNAME. The factory will create instances of the test classes then launch the simulation.

7.2 Launching Simulations with UVM

We will see a trend with the UVM going forward. That trend is automating the creation of the object-oriented testbench. We start with seeing how the UVM automatically creates the equivalent of the top-level testbench class from our OO testbench of last section.

Instead of instantiating our testbench class then calling the execute task, we now let the UVM do the work:

```
module top;
    //These come from the UVM
    import uvm_pkg::*;
    `include "uvm_macros.svh"

    //These are supplied by us
    import tinyalu_pkg::*;
    `include "tinyalu_macros.svh"

    tinyalu_bfm bfm();
    tinyalu DUT (bfm.dest);

    initial begin
        uvm_config_db #(virtual interface tinyalu_bfm)::set(null, "*", "bfm", bfm);
        run_test();
    end
endmodule : top
```

Alright, we still have to the same two things to let UVM do the work: store a handle to the BFM and call the run test method.

Remember how I said there were multiple approaches to passing an interface handle in the last section? The above code features a different approach. Here, we can't pass our interface handle via a constructor because the UVM requires specific arguments for its own constructor. Instead of exposing the constructor, we use the uvm_config_db class. The set method of this class allows us to make the

BFM available across the entire testbench. The arguments null and "*" make the data available across the entire testbench. The third argument supplies the string that names the data we're storing in the database. The fourth argument is the value being stored, the handle to the bfm in our case.

The invocation of the run test method will read the +UVM_TESTNAME parameter and use the UVM factory to instantiate a object of the test class name, whose definition is provided in our own package.

7.3 Defining and Registering a UVM Test

We need to define the test classes such that they are compatible with what the UVM expects to see.

```
class random_test extends uvm_test;
    `uvm_component_utils(random_test); //register test name in UVM factory

    virtual interface tinyalu_bfm bfm; //variable to store handle to bfm

    function new (string name, uvm_component parent);
        //Call parent constructor
        super.new(name, parent);
        //Grab bfm handle from database
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
    endfunction : new
endclass
```

A couple things to note. First, our random test class extends the uvm_class which extends the uvm_component class. In order to satisfy our base class constructor, we need to pass it a name of type string and a parent of type uvm_component (these are explained later). Second, we invoke the get method from the uvm_config_db class to retrieve our class handle and store it in the bfm variable. Finally, the `uvm_component_utils macro registers the random test class with the factory and allows us to generate objects of this type.

7.4 The Run Phase Method

The test object created from the UVM factory will inherit a run_phase method. After being created, the UVM will invoke this run phase method to execute our test. As a result, we need to override the run phase method in our test class. Some strict rules on this: the task must be called run_phase and it must have a single arugment of type uvm_phase called phase.

```
task run_phase(uvm_phase phase);
    random_tester random_tester_h;
    coverage coverage_h;
    scoreboard scoreboard_h;

    phase.raise_objection(this);

    random_tester_h = new(bfm);
    coverage_h = new(bfm);
    scoreboard_h = new(bfm);

    fork
        coverage_h.execute();
        scoreboard_h.execute();
    join_none

    random_tester_h.execute();
    phase.drop_objection(this);
endtask : run_phase
```

This method will be invoked by the UVM after our random test class object is created.

The phase object invokes a method called raise objection to indicate that we (this in OOP speak) are currently running our test and would prefer in the over-arching simulation was not stopped. By invoking the drop objection method, we are saying that (this) thread no longer objects to the simulation stopping. After all objections are resolved, we can safely stop the simulation.

7.5 Extending to Several Test Classes

Any additional test class we want to add follow the same formula:

```
class nop_test extends uvm_test;
    `uvm_component_utils(nop_test); //register test name in UVM factory

    virtual interface tinyalu_bfm bfm; //variable to store handle to bfm

    function new (string name, uvm_component parent);
        //Call parent constructor
        super.new(name, parent);
        //Grab bfm handle from database
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
    endfunction : new

    task run_phase(uvm_phase phase);
        nop_tester nop_tester_h; //Create handle for tester object
        //rest of test
    endtask
endclass
```

8 UVM Components

The UVM component class is the building block of the structure of a UVM testbench. As a result, we are often required to define and instantiate classes that extend the UVM component class, like we did for the test classes in the last section. There is a four step process to make any generic class into a UVM component.

- Extend the uvm_component class or child class to define your component
- Use the `uvm_component_utils() macro to register the class in the UVM factory
- Provide at least the minimum uvm_component constructor arguments
- Override the UVM phase methods as necessary

We illustrate this process by re-casting the tester, coverage, and scoreboard classes from the last section as UVM components.

```
class scoreboard extends uvm_component; //Req. 1
    `uvm_component_utils(scoreboard); //Req. 2

    virtual interface tinyalu_bfm bfm;

    function new(string name, uvm_component parent);
        super.new(name, parent); //Req. 3
    endfunction : new
endclass : scoreboard
```

You will notice that requirement 4 is missing from the above code. Every UVM component will inherit many phases from the base class. The UVM will step through these phases over the course of the

simulation lifetime. You can customize your component by overriding a specific phase in the derived class. All phase methods take one argument: phase of type uvm_phase. The phase methods do not need to be overridden, and when you do, you should call super.¡phase method¿ to ensure your not ignoring vital code. A couple phase method examples include:

- build_phase - UVM builds testbench hierarchy from top down. UVM components are instantiated here.
- connect_phase - UVM connects components together.
- end_of_elaboration_phase - UVM calls this after all components are connected.
- run_phase - UVM calls this task in its own thread. All run_phase methods will concurrently.
- report_phase - Runs after last objection is dropped and shows stats.

We first override the build phase in the scoreboard class:

```
function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
        $fatal();
function : build_phase
```

By retrieving the bfm from the database, we no longer have to pass it into the class in the top-level.

Second, we override the run phase method. Note that run phase is the only phase method that is a task, the rest are functions. Run phase must be a task becomes it can consume simulation time if waiting for a clock edge.

```
task run_phase(uvm_phase phase);
    shortint pred_result;
    forever begin : check_loop
        @(posedge bfm.done)
            case(bfm.op)
        //calculate predicted
        //Compare to actual
    end : check_loop
endtask : run_phase
```

8.1 Building a Testbench

The test classes we created in the last section can actually be simplified using the transition to UVCs and the newly introduced phase methods:

```
class random_test extends uvm_test;
    `uvm_component_utils(random_test); //register test name in UVM factory

    virtual interface tinyalu_bfm bfm; //variable to store handle to bfm

    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function build_phase (uvm_phase phase);
        //use uvm_component constructors
        tester_h = new("tester_h", this);
        coverage_h = new("coverage_h", this);
        scoreboard_h = new("scoreboard_h", this);
        if(!uvm_config_db#(virtual interface tinyalu_bfm)::get(null,"*","bfm",bfm))
            $fatal();
```

```
    endfunction
endclass

//Inherits build phase and bfm
class add_test extends random_test;
    `uvm_component_utils(add_test);
    add_tester tester_h; //override tester class

    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass
```

9 UVM Environments

Using the factory to create a top-level uvm test then having the test instantiating uvm components can lead to issues with re-usability. We want to target a solution that uses adaptable programming, where the testbench grows stronger every time we add a feature to it. This is in contrast to intractable programming, where expanding the testbench makes using it more complicated.

Adaptable coding has three guidelines:

- Create classes that do one thing very well and connect them to build a solution
- Avoid hardcoding anything if possible
- Program around interfaces, not making assumptions about the underlying implementation

Currently, our test class objects violate the first guideline because they both create the testbench structure by instantiating components and modify the testbench behavior by declaring different types of tester classes. We wish to move the testbench structure portion to the uvm environment class, uvm_env.

Before we do this, let's look at an example of an adaptable testbench. I'm gonna attempt to switch to UML-like diagrams here. First, we see the intractable solution:

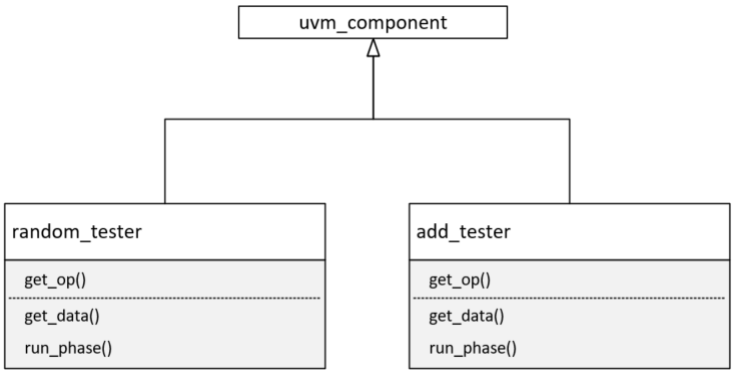

Figure 1: An intractable testbench

There are several issues with the testbench in Figure 1. A different copy of the run phase method exists in each class. Meaning that if we wanted to alter this method, we would have to update it individually in all classes that use it. This lends to very poor scaling abilities.

Now, let's come up with an adaptable version of this testbench. The first thing we notice is that the run phase method is copied in each class. This means we can move it up a level of the class hierarchy and let the child classes inherit it instead, giving us one central definition of the run phase method. The same can be said about get op and get data methods. In both classes, get data returns constrained random data to send over. In contrast, the get op method will return a random operation

in the random tester and an add operation in the add tester. This indicates that we should keep the two classes separate. However, we can think of the add operation as a subset of all possible operations from the random tester. As a result, it is most intuitive to make the add tester a child class of the random tester that will inherit the get data method and override the get op method. This is shown in UML diagram form below:

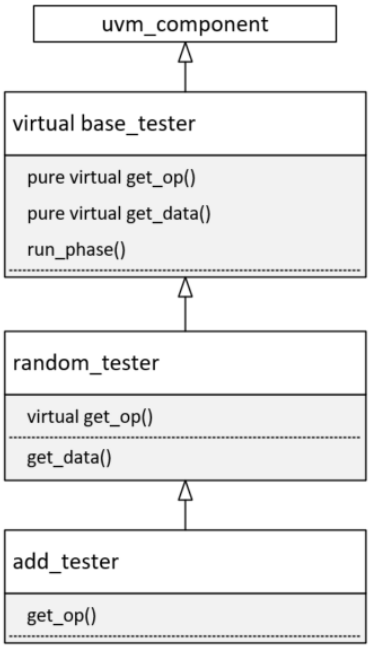

Figure 2: An adaptable testbench

The adaptability of the testbench in Figure 2 can best be shown by thinking about the effort that would be required to add a new test class, call it xor tester:

```
class xor_tester extends random_tester;
    `uvm_component_utils(xor_tester);

    function logic [2:0] get_op();
        return xor_op;
    endfunction : get_op

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass : xor_tester
```

This test class will exercise the xor functionality of the ALU. It is a very simple class definition because it has inherited the run phase method and the get data method.

9.1 Separating Structure from Stimulus

Think back to the above random test and add test classes. In the random test class, we defined the structure of the testbench as a random tester, coverage, and scoreboard. Every test class derived from the random test class must now use this specific testbench makeup. We can avoid this limitation by moving the testbench structure out of the random test class and into the UVM environment class. The uvm_env class normally only contains the build phase and connect phase methods.

The uvm_env class holds the structure of the testbench, the we use the test to specify what objects will fill in the structure. Think of it like a block game that a baby plays. The structure of the testbench determines what the shape of the holes are. The test class will specify what color block goes in each

hole. So it is up to the test class to decide if a red square or a blue square is used, but the presence of a square hole is decided by the environment class. Here is the code to define the env class:

```
class env extends uvm_env;
    `uvm_component_utils(env);

    //Create handles for the three UVCs
    base_tester tester_h;
    coverage coverage_h;
    scoreboard scoreboard_h;

    //Instantiate the UVCs
    function void build_phase (uvm_phase phase);
        tester_h = build_tester::type_id::create("tester_h",this);
        coverage_h = coverage::type_id::create("coverage_h",this);
        scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
    endfunction : build_phase

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
endclass : env
```

9.2 Creating UVCs with the UVM Factory

There are some new lines in the above code that seem complex. These are necessary to utilize the UVM factory to create a UVC. The UVM factory is better at making UVCs than our first attempt at a factory was. It is able to return the correct type of object, so we no longer have to cast the object back to the desired type. All we need to do to create a UVC with the UVM factory is the following:

```
<UVC class variable> = <UVC class name>::type_id::create("<UVC class variable>",this);
```

Invoking this static method will spit out compile time errors if you forget to register the UVC, define the UVC, or misspell the UVC.

One lingering sticking point from the above env class is the instantiation of the tester. A sharp eye will notice that it seems like we are instantiating the build tester class, however; this class is an abstract class, which can't be instantiated. The UVM factory allows this because it is making the assumption that we will override the base tester class in the factory with a child class of the base tester class before build phase is called. This is called a factory override.

A factory override takes the form:

```
<base_UVC_name>::type_id::set_type_override(<child_UVC_name>::get_type());
```

Invoking this static method tells the factory that any time it sees a base class, it needs to replace it with the specified child class. As a result, we can now rewrite the random test class:

```
class random_test extends uvm_test;
    `uvm_component_utils(random_test);
    env env_h;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function build_phase(uvm_phase phase);
        //override type of object the factory creates for the base tester
        base_tester::type_id::set_type_override(random_tester::get_type());
        //instantiate the env class
        env_h = env::type_id::create("env_h",this);
```

```
    endfunction : build_phase

endclass
```

All tests will override the factory with the tester that they need. Meanwhile, the env class will handle setting up the environment and kicking off the objects. We have officially separated the testbench structure from the test class. All that needs to be done by the test class now is to plug in the correct stimulus or tester type.